Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Copyright 1999, Analog Devices, Inc. All rights reserved. Analog Devices assumes no responsibility for customer product design or the use or application of customers' products or for any infringements of patents or rights of others which may result from Analog Devices assistance. All trademarks and logos are property of their respective holders. Information furnished by Analog Devices Applications and Development Tools Engineers is believed to be accurate and reliable, however no responsibility is assumed by Analog Devices regarding the technical accuracy of the content provided in all Analog Devices' Engineer-to-Engineer Notes.

ADSP-2100 Family C Programming Examples Guide

A compilation of C tips, DSPatch articles, Q & A's and EE-Note programming examples from ADI's DSP Applications and Development Tools Groups

ver 1.0

Edited By: JT
ADI DSP Applications

1. "*C programming for DSP*" DSPatch Reprints

This reprint of "C Programming For DSP" articles from DSPatch #26-33 covers topics on basics to get started, C runtime environments, using in-link assembly code, using library functions, and ends with an FFT example that will run on an EZ-LAB board.

The following article reprints in this section provide helpful hints and recommendations for programming DSP routines in C. Particular issues dealing with writing efficient C programs which take full advantage of the ADSP-2100 and ASDP-21000 family architectures will be presented. This article addresses arguments for using C as well as arguments for not using C, while future articles will address implementation issues.

1.1 Introduction (#26)

The C programming language has found wide acceptance as the primary programming language for embedded microprocessor software and firmware development. The demonstrated utility of C in embedded applications development can be applied to DSP applications as well. The arguments for
this are compelling.

- Many engineers and most newly graduated engineers know C already, so the development of applications can start quickly, without the burden of learning the eccentricities of a new assembly language and instruction set of the particular DSP chip the engineer may be using.

- C allows for applications to be developed, prototyped, analyzed and tested in a workstation or PC environment using commercially available c compilers and tools. This prototyped code can be used, sometimes unaltered, as the basis for the application software running on the DSP target.

- C is well defined. Therefore, C code should be portable to any DSP chip for which there exists C language support. The stability of C in the future is insured with the adoption of the ANSI standard for C.

- The majority of the software developed for DSP applications is really control code requiring sophisticated data structures and complicated control flows. This control code typically represents a small fraction of application execution time, but a significant portion of the software engineering effort. Control code is ideally suited for development in C. Also, C allows easy access to the underlying hardware. Code that needs to access hardware control and status registers and I/O ports can be written in C. Using C is easier than assembly language for control code and will decrease the amount of time is takes to develop an application.

Argument Against C

Despite the arguments put forth above, the track record of C as a programming language for DSP applications development has not been that good.

In the early eighties, soon after single chip DSP applications became practical, users began to demand more sophistication from DSP code development tools. In particular, users were demanding C support. Vendors responded with C compilers. These first generation C compilers often did not take advantage of the powerful

looping and array addressing features found in DSP architectures, and in general were too inefficient for the performance demands of DSP applications.

Topics Shown In the Following Sections

In the following sections, we will discuss elements of C programs and use of DSP/C to help make your code as efficient as possible. Topics of discussion will include loops, data arrays, use of in-line assembly code, and parameter passing. If there is something special that you would like to see addressed, drop us a line at 781-461-3672

1.2 Initializing Memory-Mapped Control Registers On The ADSP-2101 (#27)

On-chip peripherals of the ADSP-2100 family processors such as serial ports, timer, and host interface port are configured with memory-mapped registers located in the high 1K of internal data memory. To set up these control registers, your C program must write initialization values to them after system powerup.

Instead of resorting to assembly language (by writing inline code, or by linking with an external assembly code module), you can write to the memory-mapped control registers directly from C. The following ADSP-2101 example program shows how this can be done.

The main program is contained in `example.c`, shown in Listing 1. This program outputs a count through SPORT0 (serial port 0). The header file `cdef2101.h`, shown in Listing 2, uses the `#define` preprocessor directive to assign a symbolic name to each memory-mapped register.

```
/* example.c*/
#include "header.h"
#pragma inline_data
.VAR/DM  loop_count_;
#pragma inline_data
extern ushort loop_count;
#define CONSTANT 0
void init2101();
void transmit(short i);

/* This example program shows how to initialize */
/* ADSP-21xx memory-mapped registers with the   */
/* init2101() function. It also shows how the   */
/* transmit() function is used to transmit data */
/* through a serial port.                       */

main()
{
  short i;
  init2101();       /* Initialize mem-mapped regs, */
                /* including those for SPORT0. */
  loop_count=5;     /* Set up loop counter         */

  for(i=CONSTANT; i<loop_count; i++)
```

```
    {
      transmit(i);  /* Call inline assembly function */
                /* to transmit i out SPORT0.     */
    }
}

void transmit(short i)
{
  #pragma inline
  #include "asm_sprt.h"
  readstack(SI,1,DM);   /* Read i value from C stack */
  TX0=SI;               /* Transmit                  */
  #pragma inline
}
```

Listing 1 example.c

```
/* cdef2101.h */

/* This is the header file that assigns the ADSP-2101's */
/* memory-mapped registers to user-defined names.       */

#define sp1_autobuf               *(int *)        0x3fef
#define sp1_rfsdiv                *(int *)        0x3ff0
#define sp1_sclkdiv               *(int *)        0x3ff1
#define sp1_control_reg           *(int *)        0x3ff2
#define sp0_autobuf               *(int *)        0x3ff3
#define sp0_rfsdiv                *(int *)        0x3ff4
#define sp0_sclkdiv               *(int *)        0x3ff5
#define sp0_control_reg           *(int *)        0x3ff6
#define sp0_multiTX0              *(int *)        0x3ff7
#define sp0_multiTX1              *(int *)        0x3ff8
#define sp0_multiRX0              *(int *)        0x3ff9
#define sp0_multiRX1              *(int *)        0x3ffa
#define tscale                    *(int *)        0x3ffb
#define tcount                    *(int *)        0x3ffc
#define tperiod                   *(int *)        0x3ffd
#define dm_wait_reg               *(int *)        0x3ffe
#define system_control_reg        *(int *)        0x3fff
```

Listing 2 cdef2101.h

The preprocessor directive:

```
#define system_control_reg  *(int *) 0x3fff
```

casts 0x3FFF as a pointer to an integer, and then dereferences it. In other words, the symbol `system_control_reg` becomes equivalent to the integer value pointed to by 0x3FFF. It can now be used in a C statement such as

```
system_control_reg=0x1000;
```

which writes the value 0x1000 to the memory-mapped location 0x3FFF. This C statement compiles to the following assembly language instructions:

```
IO=0X3FFF;
DM(IO,M2)=0X1000;      /*M2=0*/
```

The memory-mapped registers are initialized by calling the function `init2101(),` which uses the symbols defined in `cdef2101.h` .

The `init2101()` function does not take any arguments or return a value. In `example.c`, we call this function with the statement:

```
init2101();
```

The function is contained in a file external to the main program, `init2101.c` (shown in Listing 3), which must be linked with the main program. This is most easily done by including it on the command line when you invoke the compiler:

```
/* init2101.c */

/* Here is a C routine that will initialize */
/* the ADSP-2101's memory-mapped registers. */

#include "cdef2101.h"
void init2101()
{
      dm_wait_reg          = 0x0;     /* No wait states */
      sp1_autobuf          = 0x0;
      sp1_rfsdiv           = 0x0;
      sp1_sclkdiv          = 0x0;
      sp1_control_reg      = 0x0;
      sp0_autobuf          = 0x0;
      sp0_rfsdiv           = 0x255;   /* Divide by 256 for 8kHz sampling */
      sp0_sclkdiv          = 0x2;     /* Generate 2.048 MHz SCLK         */
      sp0_control_reg      = 0x6b27;  /* Internal serial clock,          */
                              /* Rec frame sync rqd, normal      */
                              /* Trans frame sync rqd, normal    */
                              /* Int trans, rec frame sync ena   */
                              /* Mu-law compand, 8-bit word      */
      sp0_multiTX0         = 0x0;
      sp0_multiTX1         = 0x0;
      sp0_multiRX0         = 0x0;
      sp0_multiRX1         = 0x0;
      tscale               = 0x0;
      tcount               = 0x0;
      tperiod              = 0x0;
      system_control_reg   = 0x1000;  /* SPORT0 enabled,     */
                              /* SPORT1 = IRQ1, IRQ0 */
}
```

Listing 3 init2101.h

```
> CC21 example init2101
```

1.3 Getting Started (#28)

In this section we will focus on the basics associated with getting started in writing a well structured program. Before writing your program it is useful to understand the differences in performance between a program written in C and a program written in assembly language. It is also necessary to understand the issues related to writing an algorithm for an embedded DSP system in C.

It is recommended that you be familiar with the runtime model for the C compiler and be constantly aware of the ADSP-21xx registers that are used to avoid a conflict.

Evaluating Assembly Vs. C

With every embedded real-time DSP design comes the question of programming language. When using an Analog Devices' DSP you have two choices: C or assembly language. There are several trade-offs to consider when choosing between assembly language, C, or a combination of the two. These include throughput and memory requirements, development schedules, portability, the existence of pre-written code, and the experience of the programmers.

The first thing to realize is that coding in assembly language will always be more efficient and take less program memory space then code written in C. If you're not writing code for a real-time application, throughput may not be a concern (although available memory may still be). The efficiency between C code and assembly could be on the order of three to one. However, there are still many valid reasons to choose C as your programming language.

If you already have a substantial amount of code written in C for your application, you obviously want to make full use of the Analog Devices' C compiler. If you are concerned with the portability of the code between different platforms, you will want to consider writing the majority of your code in C. Analog Devices' assembly language will only work on an Analog Devices' DSP and there are no efficient conversion programs available to translate code between different manufacturer's DSPs.

The experience of the programmer(s) and your design schedule will have a large impact on your assembly vs. C decision. Although the Analog Devices' assembly language is one of the easiest assembly languages to learn and program with, you are more likely to have programmers who are already knowledgeable in the C language.

Thus if you are facing aggressive design schedules you may want to program in C to save yourself the learning curve of becoming proficient in a new language.

It's important to remember that writing in C will be faster only if you don't have to spend many hours optimizing your code to fit the memory and throughput requirements of your system. You'll want to have a good "feel" for these requirements before you start coding.

If you are going to be throughput limited or memory limited, the first thing you want to check is the Analog Devices' C library. The library is a collection of functions written in assembly code that can be called from your C language programs. Common functions include `math.h` files (`cos, sin, sqrt`, etc.), filter routines, FFTs, etc. This is your first step to optimizing your code.

To further optimize your C code, you could choose to mix assembly and C code.

The most efficient way to do this is to identify where the bottlenecks are in your code. This could be done by profiling your code with the simulator. Take these sections of C code and break them out into a callable function. This function can be in a completely different file/module, or it can be a subfunction called within a file/module. If you're selective about which functions you choose, you'll find that coding just a few routines in assembly will significantly improve your throughput. When mixing C and assembly code, the extended-assembler feature of the C compiler allows you to reference C variables without worrying about where they are located.

The compiler may attempt to move the assembly instructions to generate more efficient C code. The ***volatile*** qualifier will inform the compiler that these instructions should not be moved.

Embedded System Concerns

If you're accustomed to writing C code for a computer platform such as a PC, you can take advantage of such features as standard I/O and virtual memory. If you're writing code for an embedded application, such as a DSP processor, you have to be a lot more concerned with the I/O structure of the processor, its' interrupt structure, the register set, how it handles a C stack, and the amount of program and data memory available to you.

Standard I/O functions are contained in the stdio.h header file. Standard I/O includes character I/O (`getchar, putchar`), formatted I/O (`printf, scanf`), and file I/O. The Analog Devices' C library does not support the `stdio.h` functions since they aren't valid in an embedded application. Therefore, as a programmer for an embedded system, you'll have to be much more familiar with the I/O structure of the DSP.

A DSP's I/O could consist of any of the following: serial ports, host-interface ports (HIPs), on-board codecs, or memory-mapped A/Ds and D/As. Regardless of what type of I/O is on the DSP you'll be accessing it in one of two ways. You'll either be accessing a memory-mapped register (which you can do in C) or accessing an internal DSP register (which you'll need to do with in-line assembly).

Because there are no formatted I/O functions available to you in an embedded system, there are no `printf` functions (which allows you to print to a computer screen) that many programmers use to help debug their code. If you're transferring existing C code to an embedded application that has these types of functions you can either comment them all out or create a dummy `printf()` function that will replace the `stdio.h` functions.

The memory space of a DSP is based on a modified Harvard architecture, as opposed to the Von Neumann architecture of microprocessors and microcontrollers. In a Von Neumann architecture the program instructions and data are all stored in one common memory space. In a Harvard architecture, there are separate program and data memory spaces. With a "modified" Harvard architecture, you can store program memory data (such as filter coefficients) in program memory as well as program instructions. Why is this important? You'll want to consult the ADSP-2100 Family C Manual to learn how to access PM data locations so you can take advantage of the DSP's architecture where two data words and
an instruction can be fetched in a single cycle.

Many of the features that are needed for an embedded system are handled by the runtime header. The runtime header sets up the interrupt vector table, performs needed variable and processor-specific parameter declarations, sets up the C stack and heap, and calls the main routine. Analog Devices' C

compiler will automatically add a run-time header to your code, but you should be familiar with its contents. You may want to customize this runtime header during the course of your coding effort. For more information refer to the *ADSP-2100 Family C Compiler* and *Runtime Library Manual*.

21xx C Compiler Register Usage

The C compiler creates code that uses certain registers. You should understand how the C compiler uses registers to avoid a conflict.

A function that returns a one-word type (`int`, `char`, `short` and one-word structures) will place the return value in AR.

Functions that return a two-word type (`float`, `double`, two-word structures) will return the MSW in SR1 and the LSW in SR0.

The following scratch registers do not need to be saved by a function:

AR, AF, AY1, MR2, MR1, MR0, MF, MY1, I1,
M3, I6, M5, SB, SE, SI, SR1, SR0, PX

Parameter Passing When Calling Functions

The compiler passes parameters to functions in registers whenever possible. The compiler uses the following rules to determine how to pass parameters to functions.

- The first single word parameter is passed in **AR** (except when #4 is true).
- The second single word parameter is passed in **AY1** (except when #4 is true).
- All other parameters are passed on the stack.
- If the function is prototyped with varargs, the last named argument and all variable arguments will be passed on the stack.
- Multi-word parameters and all parameters that follow the multi-word parameter will be passed on the stack.

21xx Compiler Fixed Value Registers (Don't Touch!)

The following registers have fixed values. These registers may not be modified by the user's code.

M1 == 1

M2 == 0

I4 == stack pointer (This may be modified during stack management.)

M4 == frame pointer (This may be modified during stack management.)

M6 == 0

The L-registers should be zero in the C environment, but they may be temporarily modified by user's code, as long as they are returned to zero. The interrupt dispatcher will set all L-registers to zero, except L2 and L3 before calling the interrupt service routine.

The L2 and L3 registers are not altered by the interrupt dispatcher because they are often used as autobuffer registers, and altering them may cause incorrect operation of the autobuffer hardware.

The C Runtime Library does not use the I2-L2 or I3-L3 registers. The user must use caution when setting L2 or L3 to a value other than zero while reserving I2 and I3.

User Registers

The DAG registers I2 and I3 may be reserved. When the user reserves these registers, they are not used by the compiler. The C Runtime Library does not use the I2 or I3 registers. These two registers are often used for autobuffering applications. The user can reserve these registers and they can be modified with functions in the runtime library.

If the user does not reserve I2 or I3, then the L2 and L3 registers should be used with extreme caution.

Using The Loader Option Of The PROM Splitter

The ADSP-2100 family processors provide eight boot pages which are software selectable. This allows you to have up to eight separate programs which can be booted into internal memory.

Most C programs use both internal and external memory. Single programs which need external memory or initialized data memory can be made using the -loader option of the PROM splitter.

The PROM splitter will create load code in boot pages which will initialize your data and program memories. The last boot page will contain your internal program memory image. The `-loader` does not use any of your program or data memory.

To use the `-loader` option of the PROM splitter:

- Do not put any code in boot memory.
- Link an image (`.exe` file) which contains only program and data memory. Feel free to use initialized data memory and external memory.

- Invoke the PROM splitter (`spl21`) with the `-loader` option. The PROM splitter will create a bootable image which will first load, and then run your program.

These basic guidelines will help you create a program using both C language and assembly language code that you can get up and running on your DSP-based system with a minimal amount of problems. More detail on topics mentioned can be found in the C Compiler User's Manual.

In the next section, we will discuss techniques for customizing your runtime environment. Other sections will offer specific examples.

1.3 Custom 21xx Runtime Environments (#29)

In the last section, we focused on the basics associated with getting started in writing a well-structured program and discussed the differences between writing C and assembly code. In this section we'll take what we learned and walk through a simple example program from start to finish. Along the way we'll discuss how and when to customize your runtime environment.

Note: We assume that you have received and installed at least version 5.1 of the ADSP-2100 family development software. This upgrade contains the new G21 C-compiler that replaces the CC21 compiler. See the box on page 12 for important differences between CC21 to G21. If you have not installed this software yet, please install it now. If you have not received the upgrade and have previously registered your software, call DSP applications at (781) 461-3672.

An Example C Program

Let's start by creating a simple C program. The program `simple.c` initializes two variables and then calls a subroutine to add them together. Listing 4 shows this program.

```
/* simple C program */

int dm x_variable;
int pm y_variable;
int sum;
main()
      {
          x_variable=1;              /* init x value */
          y_variable=5;              /* init y value */
                                /* call add_routine */
          sum=add_routine(x_variable,y_variable);
          return(0);                 /* end of main */
      }
int add_routine(x,y)                   /* sub_routine to sum */
int x;
int y;
      {
```

```
        int z;
        z=x+y;                  /* perform addition */
        return(z);              /* return sum to main */
}
```

Listing 4 Program Listing for simple.c

We'll place one variable in data memory data space, and the other in program memory data space. Analog Devices' DSPs have the dual memory space of the Modified Harvard Architecture (program memory can store both data and opcodes and data memory stores data). This provides the advantage of being able to fetch both data words in the same instruction cycle as opposed to requiring two sequential fetches. Most microprocessors are based on the von Neumann architecture and have a common memory space for both data and code.

The G21 compiler has extensions to the ANSI language to support the DSP's dual memory space. Two keywords (`dm` and `pm`) are reserved and used with data type declarations. The variable declaration for sum does not use the dm or `pm` keyword in storage defaults to data memory (`dm`).

The program initializes the global variables `x_variable` in data memory and `y_variable` in program memory, then passes the values to `add_routine()`. The subroutine `add_routine()` takes the passed variables and returns the sum to `main()` . The local variables in the `add_routine()` are stored on the stack.

Now that we have our program, let's consider what the C compiler must do to convert these instructions to machine code for an ADSP-2100 family processor. First we must define the system configuration of our simple system. We'll use the file `simple.sys` , as shown in Listing 5, to define an ADSP-2105 system configuration using only the internal memory of the processor. A segment of boot memory ROM with a length of 1024 words is declared followed by a 1024 word program memory RAM segment (containing both code and data) and a 512 word data memory RAM segment.

```
.system simple_ach;
.adsp2105;
.mmap0;
.seg/rom/boot=0  boot_page_0[1024];
.seg/pm/ram/abs=0/code/data int_pm[1024];
.seg/dm/ram/abs=14336/data int_dm[512];
.endsys;
```

Listing 5 Program Listing for simple.sys

Use the command

`bld21 simple.sys`

to create the architecture file `simple.ach` for the G21 compiler.

The G21 command controls the operations of other software development tools as it performs the translation process (which could include C preprocessing, compiling, assembling, and/or linking). We'll use the `-v` (verbose) switch to illustrate the processes being controlled by G21 and the `-a` switch to identify our architecture file.

The command

```
g21 simple.c -v -a simple.ach
```

compiles, assembles, and links our `simple.c` file and creates the executable file `simple.exe` . It also performs some needed operations that are at first transparent to the programmer. These include adding a runtime header and allocating memory for the stack and heap.

The Runtime Header

The runtime header contains assembly code stored in the CRTL (C Runtime Library) that the G21 compiler adds to our program `simple.c` . The runtime header starts at PM location 0x0000 and contains the interrupt vector table to handle the DSP's interrupts. The runtime header also sets up the C Runtime Environment, calls the `main()` routine of our program, and has a trap routine to follow a `return()` from `main()` .

Let's take a look at the runtime header for the ADSP-2105, shown in Listing 6.

```
.MODULE/ABS=0         ADSP-2105_Runtime_Header;

.ENTRY                ___lib_prog_term;

.EXTERNAL             ___lib_setup_everything;
.EXTERNAL             main_;

.EXTERNAL             ___lib_int2_ctrl, ___lib_sp0x_ctrl, ___lib_sp0r_ctrl;
.EXTERNAL             ___lib_int1_ctrl, ___lib_int0_ctrl, ___lib_tmri_ctrl;

__Reset_vector:             CALL ___lib_setup_everything;
                      CALL main_;    {Begin C program}
___lib_prog_term:           JUMP ___lib_prog_term;
                      NOP;

__Interrupt2:               JUMP ___lib_int2_ctrl;NOP;NOP;NOP;
__Sport0_trans:             JUMP ___lib_sp0x_ctrl;NOP;NOP;NOP;
__Sport0_recv:              JUMP ___lib_sp0r_ctrl;NOP;NOP;NOP;
__Interrupt1:               JUMP ___lib_int1_ctrl;NOP;NOP;NOP;
__Interrupt0:               JUMP ___lib_int0_ctrl;NOP;NOP;NOP;
__Timer_interrupt:          JUMP ___lib_tmri_ctrl;NOP;NOP;NOP;
```

```
.ENDMOD
```

Listing 6 Program Listing for $ADI_DSP/lib/2105_hdr.dsp

On reset, the DSP begins executing code at PM[0x0000], the reset interrupt vector within the runtime header. The runtime header calls the initialization routine `_____lib_setup_everything` that sets up the processor defaults, creates the stack and heap, and handles any command line arguments.

The runtime header then calls the routine `main()` and execution of our `simple.c` program begins. If we had enabled (unmasked) an interrupt and it had occurred, the program counter would jump to the appropriate interrupt specific location in the interrupt vector table of the runtime header. The runtime header listing shows the interrupt vectors for IRQ0, IRQ1, and IRQ2 (the external interrupts of the ADSP-2105), the serial port and the timer. For example, a timer interrupt would jump to PM location 0x18 and the routine that you installed (with the CRTL `interrupt()` function) for timer interrupts would be executed.

When the return instruction in the main routine executes, the runtime header traps the code by performing a jump to itself at the label `_____lib_prog_term`.

The above runtime header is the default used by the G21 compiler. It is possible to modify the runtime header for your specific application. One reason to do so would be to modify the interrupt vector table to insert your own assembly language interrupt service routine without incurring the CRTL `interrupt()` function overhead.

To create your own runtime header, copy the appropriate header (i.e., `2105_hdr.dsp` for an ADSP-2105 system) from the `$ADI_DSP/21xx/lib` directory to your working directory. Modify the header as needed then assemble it using the command

```
asm21 -c 2105_hdr.dsp
```

to create the `.obj` file. The compiler uses `2105_hdr.dsp` by default. To specify a different runtime header, use the `-runhdr filename` G21 command line switch. Our compiler command is now:

```
g21 simple.c -v -a simple.ach -runhdr 2105_hdr.dsp
```

The Stack

The stack is a buffer of reserved memory used by the C compiler to pass variables between routines. The amount of memory needed varies in size depending upon how many variables are pushed onto the stack. The default stack size is 1024 locations of data memory RAM. The C runtime library initializes the stack. The listing for the stack initialization routine is shown in Listing 7.

Usually the size of the stack needs modification to fit your specific application. Let's take our program `simple.c` as an example. It's obvious that 1024 locations of memory are overkill for this program. Therefore, we can copy the routine stack.dsp from `$ADI_DSP/21XX/lib/src` to our working directory, reduce the size of the stack, and compile `stack.dsp` with the `simple.c` file.

```
g21 simple.c stack.dp -v -a simple.ach -runhdr 2105_hdr.dsp
```

```
/* This module will be located in the CRTL.
If you want to change the stack size, copy this file into your home
directory and assemble and link it in explicitly.
*/

.MODULE             Stack_Declaration;

.VAR/DM/RAM         stack[1024], ____top_of_stack;
.GLOBAL             ____top_of_stack;

.endmod;
```

Listing 7 Program Listing for $ADI_DSP/lib/src/stack.dsp

The Heap

The heap, like the stack, is memory reserved for C compiler use. The heap, however, is only reserved if the program makes a call to `malloc()`. The default size for the heap is 128 locations in data memory RAM. You can modify the heap size with a process similar to the one for modifying the stack size.
Listing 8 shows the heap module.

```
/* This module will be located in the CRTL.
If you want to change the heap size, copy this file into your
working directory and assemble and link it in explicitly.
*/

.MODULE             Heap_Declaration;

.VAR/DM/RAM         ___heap[128];
.GLOBAL             ___heap;

.endmod;
```

Listing 8 Program Listing for $ADI_DSP/lib/src/heap.dsp

Testing Our `simple.c` Program

At this point, you've seen what the C compiler does for you automatically, and what you can do to modify the runtime header, the stack, and the heap. Let us now return to our original program `simple.c`. We can use the following command to compile our file:

```
g21 simple.c -g -v -a simple.ach -runhdr 2105_hdr.dsp
```

The `-g` command generates debugging information used by the CBUG of the simulator. The command to invoke the ADSP-2105 simulator and test our C file is:

```
sim2101 -a simple -e simple
```

The `-a` switch tells the simulator to use the architecture file `simple.ach`, and the `-e` switch tells the simulator to load in the executable file `simple.exe` (the output of the G21 compiler).

In the next section we will take a more in-depth look at the use of in-line assembly code in your C program. This technique will allow you to increase the efficiency of your programs.

1.4 Interfacing C & assembly routines (#30)

In the last section we discussed how to customize the C runtime environment by modifying the stack, heap, and runtime header. In this installment, we'll build on that knowledge with the best ways to use assembly language to simplify and optimize your C program. We'll also discuss how to set up serial port autobuffers in C.

An example program, main.c , demonstrates assembly language interfacing concepts (see Listing 9). Other files that we'll use for our system include: `asm_fn.dsp, init2101.c , mreg2101.h , 2101_hdr.dsp , stack.dsp ,` and `simple.sys` (which the system builder processes to build `simple.ach`). The source code for all these files are available for download on the FTP site.

```
/* This is the main program for our autobuffering in C example.  */

#include "mreg2101.h"
extern init2101();
extern int rx_buf[];           /* set up a 256 buffer in data memory
                          for autobuffering */
int add(int a, int b);         /* prototype for add subroutine */

void main()
{
      int i,j,sum,result;
      asm("i2=^rx_buf_; \
         l2=%rx_buf_;");    /* setup autobuffering pointer and
                             setup autobuffering length */

      init2101();             /* initialize 2101 registers
                         including SPRT0 autobuf regs */ asm("ifc=0x3f;
      /* clear pending interrupts */  \
           nop;             /* one cycle for ifc latency */ \
           imask=0x8;");    /* unmask SPRT0 RCV interrupt */

      while (1)               /* infinite loop for realtime system */
       {
       asm("idle;");        /* wait for interrupt */
                        /* when the interrupt occurs, we know that the
                          rx_buf has received 256 words from SPORT0 */
       sum=rx_buf[0];       /* read first word of buffer */
       for (i=1;i<LENGTH-1;i++)      /* loop to sum data in buffer */
          {
          j=rx_buf[i];
          sum=add(j,sum);   /* call add subroutine summation */
          }
       asm volatile ("tx0=%0;" :: "a" (result));
                        /* output result to SPORT0 transmit */
       }
}

/* assembly subroutine to add two numbers together */
int add(int x, int y)
   {
   asm("ar=ar+ay1;");
   }
```

Listing 9 main.c--Autobuffering Program

There are several ways to add assembly language code to a C program and several reasons to do so. One reason is to execute operations that only exist in assembly language (such as the IDLE instruction) and to access dedicated DSP registers (such as IMASK). Another reason is that, for time critical operations, assembly-language code is usually more efficient than C. You can leverage the power of assembly language by using the functions of the C Runtime Library, which are written in assembly language (we'll discuss this in the next DSPatch). You can also embed assembly language instructions in your C code and link assembly-language derived object files with your C-compiled object files.

Understanding Assembly Code Syntax & Registers

First, you must have a general understanding of ADSP-2100 family assembly language programming before you can mix assembly language with C. The ADSP-2100 Family User's Manual is the definitive reference.

Second, you must understand how the G21 C compiler reserves the internal registers of an ADSP-21xx processor (see sidebar). Simply, never change the registers used to manage the C runtime stack: I4, M4 or L4. Only use I2 and I3 for serial port autobuffering (with M1, which is set to 1 by the C compiler).

G21 RESERVED REGISTERS

These registers are reserved by the compiler. Only modify them for temporary use, then restore.

Register	***Value***
L0	0
L1	0
L5	0
L6	0
L7	0
M1	1
M2	0
M6	0

These registers are reserved by the compiler, but without a predefined value. Only modify them for temporary use, then restore.

M0	AX1
M7	AY0
MX0	I0
MX1	I5
MY0	I7
AX0	

User Reserved Registers These registers can be reserved for use in autobuffering.

Register	*Value*
I2	User Defined
I3	User Defined

C Runtime Stack Registers
These registers are used for stack management. Do not change them.

I4	Stack Pointer
M4	Frame Pointer
L4	0

Scratch Registers
These registers are for general purpose. The do not need to be saved.

Register	*Value*
AF	User Defined
AR	User Defined
AY1	User Defined
I1	User Defined
I6	User Defined
M3	User Defined
M5	User Defined
MF	User Defined
MR0	User Defined
MR1	User Defined
MR2	User Defined
MY1	User Defined
PX	User Defined
SB	User Defined
SE	User Defined
SI	User Defined
SR0	User Defined
SR1	User Defined

The DSP also has memory-mapped registers that reside in internal reserved data memory. To initialize the memory-mapped registers in C, use the header file `mreg2101.h` (see Listing 10). This file defines symbolic labels to the addresses of the registers. The register for serial port 0 autobuffering setup is located at address 0x3FF3:

```
/* This header file Defines Memory Mapped registers of 2101   */
```

```
#define  SP1_Autobuf          *(int *)        0x3fef
#define  SP1_RFSDIV           *(int *)        0x3ff0
#define  SP1_SCLKDIV          *(int *)        0x3ff1
#define  SP1_Control_Reg      *(int *)        0x3ff2
#define  SP0_Autobuf          *(int *)        0x3ff3
#define  SP0_RFSDIV           *(int *)        0x3ff4
#define  SP0_SCLKDIV          *(int *)        0x3ff5
#define  SP0_Control_Reg      *(int *)        0x3ff6
#define  SP0_MultiTX0         *(int *)        0x3ff7
#define  SP0_MultiTX1         *(int *)        0x3ff8
#define  SP0_MultiRX0         *(int *)        0x3ff9
#define  SP0_MultiRX1         *(int *)        0x3ffa
#define  TSCALE               *(int *)        0x3ffb
#define  TCOUNT               *(int *)        0x3ffc
#define  TPERIOD              *(int *)        0x3ffd
#define  DM_Wait_Reg          *(int *)        0x3ffe
#define  System_Control_Reg   *(int *)        0x3fff

#define  LENGTH 64
```

Listing 10 mreg2101.h--Header File

`#define  SP0_Autobuf  *(int *) 0x3ff3`

The program `init2101.c` (see Listing 11) sets up the memory-mapped registers. For autobuffering, we'll use I2 to hold our address pointer or index, and M1 for our step or post-modify value. Write 0x25 to address 0x3FF3 in C with the syntax:

```
/*  This C routine initializes the 2101 memory mapped control registers, sets
up SPORT0 for receive autobuffer using I2, M1 registers.  SPORT0 is
initialized for internally generated SCLK, RFS & TFS required, normal frame
sync width, internal RFS & TFS, 8-bit u_law data.  */

#include "mreg2101.h"
void init2101()
{
      SP1_Autobuf          = 0x0;
      SP1_RFSDIV           = 0x0;
      SP1_SCLKDIV          = 0x0;
      SP1_Control_Reg      = 0x0;
      SP0_Autobuf          = 0x0025;
      SP0_RFSDIV           = 255;
      SP0_SCLKDIV          = 3;
      SP0_Control_Reg      = 0x6b27;
      SP0_MultiTX0         = 0x0;
      SP0_MultiTX1         = 0x0;
      SP0_MultiRX0         = 0x0;
      SP0_MultiRX1         = 0x0;
      TSCALE               = 0x0;
      TCOUNT               = 0x0;
      TPERIOD              = 0x0;
      DM_Wait_Reg          = 0x0000;
      System_Control_Reg   = 0x1018;
}
```

Listing 11 init2101.c--Register Initialization Function

```
SP0_Autobuf = 0x0025;
```

Because we chose I2 for the pointer, we must use the L2 register to hold the length of our circular buffer. You need not and should not reserve L registers; they are reserved automatically when you reserve an I register.

Now that we've defined the autobuffering registers, we must reserve the pointer to prevent the C compiler from using it. Use the `-mreserved=i2` on the command line to reserve I2. Do not try to reserve L2 on the command line.

Now that we have our registers initialized for autobuffering, we need a buffer reserved for our serial data. A memory space used for serial port autobuffering must be defined as a circular buffer. This assures that the linker will place the buffer on a suitable boundary. The best way to define a buffer as circular is to use assembly directives. We will put the assembly directives in an assembly-language file, called `asm_fn.dsp` (Listing 12), that contains the assembly routines we need. The following directives in the `asm_fn.dsp` file define a circular buffer, `rx_buf_`:

```
.MODULE asm_fn;

.VAR/DM/RAM/CIRC   rx_buf_[64];
.GLOBAL            rx_buf_;

.entry add_;
add_:
      ar=ar+ay1;
      rts;
.endmod;
```

Listing 12 asm_fn.dsp--Assembly Language Functions

```
.VAR/DM/RAM/CIRC rx_buf_[64];
.GLOBAL        rx_buf_;
```

Adding Simple Assembly Instructions To C

To add a single line of assembly code into your C with the GNU C compiler (G21, Release 5.0 or later), use the simple form of the `asm()` construct. An embedded `IDLE` instruction with the GNU compiler looks like this:

```
asm("idle;");  /* wait for interrupt */
```

To embed more than one assembly instruction at a time, you could place the assembly instructions on the same line, or place them on separate lines (which makes your code easier to read) and use the backslash (\) to indicate

a line continuation. The backslash (\) must be the last character on the line including comments. The code below uses an `asm()` statement to initialize the autobuffer pointer (`I2`) to point to the start of the buffer `rx_buf_`, and the length register (`L2`) to contain the length of the buffer:

```
asm("i2=^rx_buf_; \
     l2=%rx_buf_;");
```

The code below clears any pending interrupts by writing 0x3F to register `IFC`, and after one cycle for IFC's latency, unmasks the serial port 0 receive interrupt by writing 0x8 to the IMASK register:

```
    asm("ifc=0x3f; /* clear pending interrupts */
\nop;              /* one cycle for ifc latency */
\imask=0x8;");     /* unmask SPRT0 RCV interrupt*/
```

Assembly Construct Template

As you can see, it's easy to add assembly instructions to your C code. The difficulty lies in adding instructions that do not interfere with the registers used by the C runtime environment. The `asm()` construct has a more complex form that gives the C compiler choices of which DSP registers to use. The template for the complex `asm()` construct is:

```
asm ("template":
"constraint" (out_ops):
"constraint" (in_ops):
"clobber");
```

See the ADSP-2100 Family C Tools Manual for full information about using the complex form of the `asm()` construct. Most applications do not need the flexiblity provided by the template--the next part of this article shows a simpler alternative.

Adding Assembly Language Modules

The best way to add assembly code to your C system is to create separate assembly modules. If they are written to comply with the C runtime environment model, object files produced by the assembler can be linked with object files produced by the C compiler.

To create an assembly subroutine that is called by your C code, you need to understand how the compiler passes arguments and returns values from functions.

When passing arguments, the compiler places the first parameter into the AR register, and the second into the AY1 register. If more than two parameters are passed, they are placed on the C stack. All 16-bit results are returned in the AR register, and 32-bit results are returned in the SR1 and SR0 registers.

Let's create a small subroutine that accepts two arguments, sum and j . The first argument, sum is passed in the AR register, while j is passed in the AY1 register. We can take advantage of this knowledge to perform an addition with the assembly instruction `AR=AR+AY1` .

Our function call looks like this:

```
     extern int add(int x, int y);      /* prototype for add
function*/
     void main()
     {
     ...
     sum=add(sum, j);
     ...
     }
```

The assemble language routine looks like this:

```
     .module asm_fn;
     ...
     /* other functions
     and declarations */
     .entry add_;
     add_:
     ar=ar+ay1;
     rts;

     .endmod
```

For larger assembly subroutines, you must store the state of your registers.

Assembly Modules That Initialize The Runtime Environment

The files `stack.dsp` (see Listing 13) and `2101_hdr.dsp` (see Listing 14) are examples of assembly modules that are assembled and linked with compiled C code object files. Copy them into your working directory. Both of these files have been modified for this example (refer to the previous C Programming column for more information on altering the stack and the runtime header). They are used to initialize the C runtime environment.

```
/*This is a modified version of the STACK.DSP file that exists in the
directory $ADI_DSP\21XX\LIB\SRC. The file has been modified to decrease
the size of the stack and copied into our working directory.  */


.MODULE         Stack_Declaration;

.VAR/DM/RAM     stack[200], ____top_of_stack;
.GLOBAL         ____top_of_stack;

.endmod;
```

Listing 13 stack.dsp--Modified Version Of The Stack Module

```
/* This is a modified version of the 2101_hdr.dsp file that exists in the
directory $ADI_DSP\21XX\LIB. It's been modified to remove the C library
call to the SPORT0 receive interrupt service routine and copied into the
working directory.  */


.MODULE/ABS=0    ADSP2101_Runtime_Header;

.ENTRY           ___lib_prog_term;

.EXTERNAL        ___lib_setup_everything;
.EXTERNAL        main_;

.EXTERNAL        ___lib_int2_ctrl, ___lib_sp0x_ctrl, ___lib_sp0r_ctrl;
.EXTERNAL        ___lib_int1_ctrl, ___lib_int0_ctrl, ___lib_tmri_ctrl;

__Reset_vector:             CALL ___lib_setup_everything;
                      CALL main_;
                      /* Begin C program */
___lib_prog_term:           JUMP ___lib_prog_term;
                      NOP;

__Interrupt2:               JUMP ___lib_int2_ctrl;NOP;NOP;NOP;
__Sport0_trans:             JUMP ___lib_sp0x_ctrl;NOP;NOP;NOP;

/*  The following line has been modified for our example */
__Sport0_recv:              RTI;NOP;NOP;NOP;
__Interrupt1:               JUMP ___lib_int1_ctrl;NOP;NOP;NOP;
__Interrupt0:               JUMP ___lib_int0_ctrl;NOP;NOP;NOP;
__Timer_interrupt:          JUMP ___lib_tmri_ctrl;NOP;NOP;NOP;


.ENDMOD;
```

Listing 14 2101_hdr.dsp Modified Runtime Header

The file `stack.dsp` has no assembly instructions; it initialize buffers using assembly directives. We've modified this module to create a stack of 200 locations.

The file `2101_hdr.dsp` is a modifed runtime header that we want to use instead of the default runtime header. You must specify the modified header with the `-runhdr 2101_hdr.dsp` switch on the command line. We've modified our runtime header to simplify the SPORT0 receive interrupt vector.

After initialization, the program `main.c` enters an infinite loop and executes the `IDLE` instruction, which puts the processor in idle mode until an interrupt occurs. When the `rx_buf` is full, a serial port 0 receive interrupt occurs. The DSP automatically jumps to the interrupt vector location, hits the return from interrupt (`RTI`) instruction and returns to the instruction after the `IDLE` .

The file `2101_hdr.dsp` also contains the call to the main function. Notice that labels and variables in C have an underscore (_) appended when used in assembly code (`main` becomes `main_`).

Since the external module has one or more assembly subroutines, all the rules about parameter passing and stack management apply.

Building The Example System

Now that we've covered the basics, all that's left is to compile our example system. First, run the system builder on the `simple.sys` file (see Listing 15) to produce the `simple.ach` file. Then invoke G21 to translate your source file into the executable. During the translation process, the C compiler actually invokes several tools in this order: C preprocessor, C compiler, assembler preprocessor, assembler, linker. Considering this, you can tell the compiler at the command line to compile your C files, assemble your assembly language modules, and link all the object files together. We'll use the following command line for this example:

```
.system simple_ach;
.adsp2101;
.mmap0;
.seg/rom/boot=0  boot_page_0[2048];
.seg/pm/ram/abs=0/code/data int_pm[2048];
.seg/dm/ram/abs=14336/data int_dm[1024];
.endsys;
```

Listing 15 simple.sys--System Builder File For Simple ADSP-2101 System

```
g21 main.c init2101.c stack.dsp asm_fn.dsp -a simple.ach -g -save-
temps -Wall -o output -map -mreserved=i2 -runhdr 2101_hdr.dsp
```

- Input files are `main.c` , `init2101.c` , and `stack.dsp`.
- `-runhdr 2101_hdr.dsp` specifies the runtime header.
- `-a simple.ach` specifies the system architecture file.
- `-mreserved=i2` reserved I2 and L2 registers for autobuffering.
- `-g -save-temps` are needed for use to use the C debugger in the simulator.

. `-Wall` echoes all possible warnings to the screen.

. `-o output -map` names the output file `output.exe` and creates the map file `output.map`.

1.5 Using C Library Functions (#31)

This section presents the uses and syntax of C library functions and macros.

Using library calls and the information gained in the last two sections we'll create an example program that receives blocks of data through serial port autobuffering, performs a Fast-Fourier Transform on the data, manipulates the data, then performs the inverse transform and outputs the result. This program provides a simple but useful example of signal processing techniques that can be run on the ADSP-2101 EZ-LAB board.

To quickly summarize, the last two C Programming columns discussed how to customize the C Runtime Environment by modifying the stack, heap, and runtime header (DSPatch #29), and explained the best ways to add assembly code to simplify and optimize your C code (DSPatch #30) using a serial port autobuffering example.

This example program was compiled with Analog Devices' software development tools, release 5.01. To determine what version of tools you have, type ASM21 -H at the command line (the C prompt for DOS) and check the release number. If you plan on doing any serious C code development using Analog Devices DSPs and don't already have release 5.x, you should consider the upgrade. Release 6.0 is the production release of the GNU based ADSP-2100 Family C Compiler (G21) and has many improvements over release 5.1 and 5.01. For a summary of the new GNU based compiler features of 5.x and 6.0 see sidebar. For upgrade information, contact Computer Products Division Customer Support at (781) 461-3881.

WHAT'S THE DIFFERENCE BETWEEN CC21 and G21?

If you've already been programming with the ADDS-21XX-BUN C compiler (CC21), here are the key differences to be aware of:

G21 (new)	**CC21 (old)**
no fract	fract type
asm ( )	#pragma inline
G21 invokes linker	user invokes linker as a separate command
IEEE floating point format	Non-standard floating point format

C Callable Library Functions

C programs depend on library functions to perform operations not provided by the language. These C callable routines are normally written in assembly language and can include memory allocation, signal processing, and mathematics functions. Not taking advantage of these functions will mean you'll have to work harder to produce less optimized code.

Let's break library functions into two categories: those provided by Analog Devices and those that are user created.

Existing Library Functions

The Analog Devices development software provides most of the basic C callable functions needed for programming your DSP. These functions, called the C Runtime Library, are detailed in the C Runtime Library Manual that you'll receive with your development software.

The C Runtime Library consists of subroutines and macros defined by the ANSI C standard and extensions to ANSI C provided by Analog Devices.

Functions that have the same purpose are grouped into header files. For a list of these header files, see sidebar.

C RUNTIME LIBRARY HEADER FILES

The following header files are uses supported by G21 C compiler

ANSI Standard Runtime Environment Macros & Definitions

Header	***Purpose***
error.h	Error Handling
stddef.h	Standard Definitions
limits.h	Limits
float.h	Floating Point

ANSI Standard Functions

Header	***Purpose***
math.h	Mathematics
signal.h	Signal Handling
stdarg.h	Variable Arguments
ctype.h	Character Handling
string.h	String Handling
stdlib.h	Standard Library
locale.h	Localization

```
assert.h        Diagnostics
setjump.h       Non-Local Jumps
stdio.h*        Input/Output
```

C Runtime Library Extensions

Header	*Purpose*
asm_sprt.h	Assembly Language Macros
circ.h	Circular/Autobuffer Routines
filters.h	DSP Filters
ffts.h	Fast Fourier Transforms
float.h	Floating Point
sport.h	Serial Port Handling

* contains only the EOF character since standard input/output has little meaning in an embedded system.

For our EZ-LAB example, we'll use functions from the header files `fft.h`, `signal.h`, and `sport.h`.

The sport.h header is new for release 5.01 software. It provides C macros for starting, stopping, reading, and writing from both serial ports on the ADSP-2101.

init2101.c program listing

```
/*  This C routine initializes the 2101 memory mapped control registers, sets
up SPORT0 for receive autobuffer using I2, M1 registers. SPORT0 is initialized
for internally generated SCLK, RFS & TFS required, normal frame sync width,
internal RFS & TFS, 8-bit u_law data. */

                #include "mreg2101.h"
                void init2101()
                {
                SP1_Autobuf           = 0x0;
                SP1_RFSDIV            = 0x0;
                SP1_SCLKDIV           = 0x0;
                SP1_Control_Reg       = 0x0;
                SP0_Autobuf           = 0x06a7;
                SP0_RFSDIV            = 255;
                SP0_SCLKDIV           = 3;
                SP0_Control_Reg       = 0x6927;
                SP0_MultiTX0          = 0x0;
                SP0_MultiTX1          = 0x0;
                SP0_MultiRX0          = 0x0;
                SP0_MultiRX1          = 0x0;
                TSCALE                = 0x0;
                TCOUNT                = 0x0;
                TPERIOD               = 0x0;
                DM_Wait_Reg           = 0x0000;
                System_Control_Reg    = 0x0;
                }
```

Listing 16 init2101.c Program Listing

To use SPORT0 and autobuffering, you need to perform the following.

1. Initialize SPORT0 control and autobuffer registers (see Listing 16, init2101.c)
2. Unmask SPORT0 receive interrupts
3. Initialize pointers for autobuffering (I2 and I3)
4. Enable SPORT0
5. Write initial value to SPORT0 transmit to begin transmit autobuffering

Steps 4 and 5 can be accomplished using macros from sport.h.

The function `sport_write()` will write a value to the specified serial ports transmit buffer.

The function `sport_start()` enables the serial port by setting the appropriate bit in the system control register.

The syntax for the `sport.h` functions is below.

```
#include <sport.h>
sport_write(0,0);
sport_start(0);
```

In order to manipulate buffers of data in C, we'll use the `memset` and `memcpy` functions in the header `string.h`. The function `memset()` allows us to initialize buffers to a constant value. This comes in handy when we want to initialize buffers to zero, as in the case of the transmit autobuffer and the imaginary time domain inputs to our FFT.

The function `memcpy()` allows us to transfer the contents of one buffer to another. We'll use the buffer rx_buf[16] to receive our autobuffered serial port inputs. We then need to transfer the contents of this buffer to the buffer `real_time_buf[16]` , which is the real time inputs to our FFT, so `rx_buf[16]` is free to receive the next block of data.

The syntax for the `string.h` functions is shown in Listing 17.

```
#include <string.h>
memset(tx_buf,'\0',N);
memset(imag_time_buf,'\0',N);
memset(real_freq_buf+N-3,'\0',N-3);
memset(imag_freq_buf+N-3,'\0',N-3);
memcpy(real_time_buf,rx_buf,16);
memcpy(tx_buf,real_time_buf,16);
```

Listing 17 Syntax For The string.h Functions

Finally, we need to use the header `fft.h` to perform our fast fourier transform and its inverse. The `fft.h` header file was omitted from the C Runtime Library Manual dated 11/93. For more information refer to the release note for release 5.01 software.

The `fft.h` header file contains transforms of size 8, 16, 32, 64, 128, 256, 512, and 1024. We've chosen a 16-point fft and its inverse for our example.

The syntax for the `ffts.h` functions is below.

```
#include <ffts.h>
fft16(real_time_buf,
     imag_time_buf,
     real_freq_buf,
     imag_freq_buf);
ifft16(real_freq_buf,
     imag_freq_buf,
     real_time_buf,
     imag_time_buf);
```

Quick DSP Sampling Theory Lesson

A serial codec (which includes both an A/D and D/A converter) digitizes inputs from a microphone plugged into the audio-in jack of the EZ-LAB board. This 8-bit codec samples the data at 8000 Hz.

Since 4 kHz is the highest frequency that can be represented from an 8 kHz sampling-frequency (as dictated by the Nyquist theory), our usable frequency range will be 0 Hz to 4000 Hz. With this codec, the actual range due to built-in filtering is 200 Hz to 3400 Hz. This still works fine for our example, since the human voice falls nicely within this range.

The DSP receives the digitized samples from the codec through the serial port. 16 samples are collected using serial port autobuffering.

These 16 inputs will be the `real_time_buf[16]` data for our 16-point FFT. We're performing a complex FFT on real data, so we initialize our imaginary input buffer `imag_time_buf[16]` to zero.

In simple terms, a FFT determines what frequencies exist in your time domain input. A 16-point FFT will divide our 0 to 4 kHz frequency range into 8 frequency "bins". The value of each 500 Hz bin will be represented by a complex number.

Increasing the number of points of your FFT will increase the number of bins and therefore the resolution within the 0 to 4 kHz range. A 1024-point FFT will divide our range into 512 bins. In this case, the resolution of the bins will be about 8 Hz. The trade-off is that a larger FFT will give you better resolution, but will also take significantly more throughput and memory locations.

A 16-point FFT was chosen for this example since we only have the internal memory of the DSP to work with on the EZ-LAB board and because it's more than adequate to illustrate the library calls and allows us to hear the effects of signal processing on the human voice.

```
2101_hdr.dsp modified runtime header file listing

/*  This is a modified version of the 2101_hdr.dsp file that
exists in the directory $ADI_DSP\21XX\LIB. It's been modified to
remove the C library call to the SPORT0 receive interrupt service
routine and copied into the working directory. */

.MODULE/ABS=0   ADSP2101_Runtime_Header;

.ENTRY     ___lib_prog_term;

.EXTERNAL  ___lib_setup_everything;
.EXTERNAL  main_;

.EXTERNAL  ___lib_int2_ctrl, ___lib_sp0x_ctrl, ___lib_sp0r_ctrl;
.EXTERNAL  ___lib_int1_ctrl, ___lib_int0_ctrl, ___lib_tmri_ctrl;

__Reset_vector:         CALL ___lib_setup_everything;
                  CALL main_;
            {Begin C program}
___lib_prog_term:       JUMP ___lib_prog_term;
                  NOP;

__Interrupt2:           JUMP ___lib_int2_ctrl;NOP;NOP;NOP;
__Sport0_trans:         JUMP ___lib_sp0x_ctrl;NOP;NOP;NOP;
/*  The following line has been modified for our example */
__Sport0_recv:          RTI;NOP;NOP;NOP;
__Interrupt1:           JUMP ___lib_int1_ctrl;NOP;NOP;NOP;
__Interrupt0:           JUMP ___lib_int0_ctrl;NOP;NOP;NOP;
__Timer_interrupt:      JUMP ___lib_tmri_ctrl;NOP;NOP;NOP;

.ENDMOD;
```

Listing 18 2101_hdr.dsp Modified Runtime Header File Listing

```
mreg2101.h header file listing
/*  This header file Defines Memory Mapped registers of 2101 */
#define  SP1_Autobuf            *(int *)        0x3fef
#define  SP1_RFSDIV             *(int *)        0x3ff0
#define  SP1_SCLKDIV            *(int *)        0x3ff1
#define  SP1_Control_Reg        *(int *)        0x3ff2
#define  SP0_Autobuf            *(int *)        0x3ff3
#define  SP0_RFSDIV             *(int *)        0x3ff4
#define  SP0_SCLKDIV            *(int *)        0x3ff5
#define  SP0_Control_Reg        *(int *)        0x3ff6
#define  SP0_MultiTX0           *(int *)        0x3ff7
#define  SP0_MultiTX1           *(int *)        0x3ff8
#define  SP0_MultiRX0           *(int *)        0x3ff9
```

```
#define  SP0_MultiRX1          *(int *)       0x3ffa
#define  TSCALE                *(int *)       0x3ffb
#define  TCOUNT                *(int *)       0x3ffc
#define  TPERIOD               *(int *)       0x3ffd
#define  DM_Wait_Reg           *(int *)       0x3ffe
#define  System_Control_Reg    *(int *)       0x3fff
#define  LENGTH 64
```

Listing 19 mreg2101.h Header File Listing

Building Our Example System With The G21 Compiler

Now that we've covered the basics, all that's left is to compile our example system. Our example program is called `main.c` (see Listing 20). Other programs that we'll use for our system include: `init2101.c`, `mreg2101.h`, `2101_hdr.dsp`, `stack.dsp`, and `ezlab.sys` (which we'll "build" to get `ezlab.ach`). All of these source files are available from Analog Devices' computer bulletin board system (BBS). See the back page of DSPatch for information on how to connect to the BBS.

```
main.c program listing
/* This is the main program for our Voice Processing EZ-LAB example.*/

#define N 16
#include "mreg2101.h"
#include <string.h>
#include <ffts.h>
#include <sport.h>

asm(".var/dm/ram/circ rx_buf_[16];  \
     .global rx_buf_;               \
     .var/dm/ram/circ tx_buf_[16];  \
     .global tx_buf_;");
int real_time_buf[N];
int imag_time_buf[N];
int real_freq_buf[N];
int imag_freq_buf[N];

extern init2101();
extern int rx_buf[];  /* set up a 16 buffer in data memory for autobuffering */
extern int tx_buf[];  /* set up a 16 buffer in data memory for autobuffering */
void main()
{

/* initialize 2101 registers including SPORT0 autobuf regs */
      init2101();

/* clear pending interrupts, set IRQs to edge sensitive and
   umask SPORT0 RCV interrupt */
      asm("ifc=0x3f;     /* clear pending interrupts     */ \
           icntl=0x7;    /* make irq ints edge sensitive */ \
           imask=0x8;"); /* unmask SPRT0 RCV interrupt   */

/* setup autobuffering pointers and setup autobuffering lengths */
      asm("i2=^rx_buf_; \
```

```
      l2=%rx_buf_;");
      asm("i3=^tx_buf_; \
      l3=%tx_buf_;");

/* initialize tx_buf[16] to all zeros */
      memset(tx_buf,'\0',N);

/* write a zero to tx0 buffer to start autobuffering */
      sport_write(0,0);

/* enable SPORT0 by setting system control register bit 12 */
      sport_start(0);

/* infinite loop for realtime system */
      while (1)
        {
      /* wait for interrupt */
        asm volatile("idle;");

      /* copy sport0 rcv buffer to fft real input buffer */
        memcpy(real_time_buf,rx_buf,16);


      /* zero fft imaginary input buffer */
        memset(imag_time_buf,'\0',N);

      /* perform 16-point complex FFT on time domain inputs */
        fft16(real_time_buf,
            imag_time_buf,
            real_freq_buf,
            imag_freq_buf);
      /* zero the upper bins of FFT results to create low pass filter */
        memset(real_freq_buf+N-3,'\0',N-3);
        memset(imag_freq_buf+N-3,'\0',N-3);

      /* perform 16-point inverse FFT on freq domain inputs */
        ifft16(real_freq_buf,
             imag_freq_buf,
             real_time_buf,
             imag_time_buf);
      /* copy output of inverse FFT to transmit buffer for autobuffering */
        memcpy(tx_buf,real_time_buf,16);
        }
}
```

Listing 20 main.c EZ-LAB Example Program Listing

Considering this, you can compile your C files, assemble your assembly modules, and link them together all on the command line. We'll use the following for this example.

```
g21 @allfiles -a ezlab.ach -g -save-temps -Wall -o output -map
-mreserved=i2,i3 -runhdr 2101_hdr.dsp
```

The ASCII file, `allfiles`, contains the list of our input files, which are `main.c`, `init2101.c`, and `stack.dsp`. The description of the compiler switches used in this example are as follows:

`-runhdr 2101_hdr.dsp` specifies the runtime header

`-a ezlab.ach` specifies the system architecture file

`-mreserved=i2,i3` reserved the register pairs I2/L2 and I3/L3 for autobuffering

`-g -save-temps` are needed to use the C debugger in the simulator

`-Wall` echos all possible warnings to the screen

`-o output -map` names the output file output.exe and creates the map file output.map

```
stack.dsp modified version of the stack program

/* This is a modified version of the STACK.DSP file that exists in the
directory $ADI_DSP\21XX\LIB\SRC. The file has been modified to decrease the
size of the stack and copied into our working directory. */

.MODULE         Stack_Declaration;
.VAR/DM/RAM     stack[300], ____top_of_stack;
.GLOBAL         ____top_of_stack;
.endmod;
```

Listing 21 stack.dsp Modified Listing

Running Our Program On The ADSP-2101 EZ-LAB

Two things are necessary before you can run this example system on an EZ-LAB board. First, you must program an EPROM. To do this, you must use the prom splitter, `SPL21`. For more information on proper use of `SPL21`, refer to the software tools manual.

Second, you need to make the following modification of your board before running the code on your EZ-LAB.

Pins 4 and 5 on the serial port connector should be tied together. This will connect Receive Frame Sync (RFS0) to Transmit Frame Sync (TFS0) and is needed for proper framing of the serial port for transmit autobuffering to work correctly. Note that pin 1 of the connector is the top right pin when facing the connector. The *ADSP-2100 Family EZ TOOLS Manual* contains all the details of the EZ-LAB operation. The pinouts for the serial port and all other connectors are listed in this manual.

All of the source files required to build and run this example application on the EZ-LAB board, can be found on the DSP Bulletin Board System. (See the back page of DSPatch for information on how to connect to the BBS.)

In the next section we will look at using standard I/O effectively.

1.6 Handling STDIO - ADSP-2171 Example

In this installment of C Programming For DSP, we will discuss how to handle STDIO in your C programs and we'll also talk about some special considerations when writing C code for one of our newest DSPs, the ADSP-2171.

`STDIO.H` is an ANSI C header file that defines types and macros needed to handle stream-level I/O for a C system. For example, the commonly used `printf` command is defined in the STDIO library.

`STDIO.H` has many uses when writing C code for a PC-based C compiler. However, in an embedded DSP-based system, there is rarely a need to print a stream of information to a monitor, or receive a stream of characters from a keyboard. Therefore, the `STDIO.H` library is not supported in the Analog Devices fixed-point software.

There is a file named `STDIO.H` included with the ADDS-21XX-SW-PC software, but it is an empty file containing only an `EOF` character.

Most programmers writing code for the ADSP-21xx family won't have a need for the STDIO library. However, there are two cases where you may find you do have STDIO functions in your code.

In the first case, you may be benchmarking a section of prewritten C code on the ADSP-21xx family and your code already has many instances of stream-level I/O (`scanf` , `printf` , etc.). You'd probably rather not manually strip out or comment all instances of stream-level I/O in your code.

In the second case, you may want to write and test your C code using a PC-based C compiler (like Borland's Turbo C++) before compiling the code with the Analog Devices development tools. You may want to use `printf` in your code to print out error messages or status messages.

In either case, you have only two choices. The first of course is to remove all instances of `printf`, etc. from your code before compiling with the Analog Devices G21 compiler. The second is to edit the blank `STDIO.H` header (or create a new header) and add the following for each stream-level I/O function that exists in your code:

```
#define printf myprintf
int myprintf (char *fmt,...)
{return 1;}
```

These modifications you have included in the new `STDIO.H` header file in your file will enable each `printf` to be ignored. The `myprintf` function is called instead of the standard printf function. If some sort of I/O handling is desired, the `myprintf` function can accomplish this. This I/O handling may consist of a voltage or series of voltages being sent out a D/A converter. After modifying `STDIO.H` you can compile and run your C program.

We took an extra step by defining `printf` as the new symbolic `myprintf`. This is only needed if you have your own `STDIO.H` header file that contains valid `printf` declarations. If you do not, you can skip the `#define` and use `printf` in the `int` line.

There are a couple of things to consider with our approach. First, since stream-level I/O is ignored, your program may not work the way you had originally intended it to. If you want to use the I/O of the DSP, you'll want to look at the header file `SPORT.H` which controls the serial ports, or create your own I/O functions.

Second, even though your `printf` is now being ignored, it is still seen as a function call. Therefore your code will still have the overhead associated with saving and restoring registers for each call. If throughput is an issue, it may be more beneficial to remove the `printf` function after all. Note: If you do not use the `-msmall-code` switch, the overhead should be a simple stack/frame overhead of a few cycles. Let's look at a very simple program to illustrate the above concept.

The program `test.c` is written for the ADSP-2171 processor. The main routine uses in-line assembly to test a bit in the AR register. We've also assured that the bit is set to zero for this example so the `error_routine` will be called.

The `error_routine` subroutine uses a `printf` to print an error message to the screen (useful for PC debugging) and outputs the hex value 0xDEAD to a memory-mapped parallel port (useful when using a logic analyzer while testing your embedded system).

The test of the bit in the main routine could easily have been written in C code. And, in fact, you probably wouldn't be using in-line assembly if you were compiling your code first using a PC-based compiler. However, we used assembly to highlight one of the new instructions available on the ADSP-2171 processor.

Because the ADSP-2171 processor has additional instructions not available on other members of the ADSP-21xx family (such as bit manipulation and X*X squaring capability--see the ADSP-2171 Data Sheet for more information), there is an additional switch needed during assembly.

The port `log_ana_port` is defined in the system file. The following instruction would be added to the system file `2171.SYS`:

```
.port/dm/abs=0x2000
log_ana_port_;
```

The underscore is added because we'll be using this label in both assembly and C.

The port then needs to be declared in an assembly module. Copy the file `2171_hdr.dsp` into your working directory from the directory `c:\adi_dsp\21xx\lib` and add the following:

```
.PORT log_ana_port_;
.GLOBAL log_ana_port_;
```

These assembly directives declare the port and make it global.

This could be done in any assembly file. The `2171_hdr.dsp` file was chosen because it is likely that you'll be modifying it anyway.

Now, the port label can be used in a C file by using the syntax:

```
extern volatile int
log_ana_port;
```

You should note that the underscore, as the last character of the symbol, is not used in the C syntax. The C Compiler, when generating assembly code, will concatenate an underscore character at the end of any symbol used in the C program. You must keep this in mind when referring to a symbol in mixed C and assembly code programs. Since the G21 compiler is an executable that can perform compiling, assembly, and linking of your code, we will perform each as an individual step.

The batch file `build.bat` contains the individual steps needed to compile a C program for the ADSP-2171 processor. The first step builds the architecture file that will be needed during linking.

The second step is to build our `2171_hdr.dsp` assembly using `asm21`.

The third step is to compile our C program. The -S (this must be an uppercase S) stops G21 before the assembly step.

Next we'll run the C preprocessor, CPP. This is needed if any C preprocessor directives are used in your assembly code like `asm ("#include<def2171.h>");`.

After the preprocessor, the assembler, ASM2, is run. Note the switch -2171. This is needed when any ADSP-2171 assembly instructions are used like TSTBIT.

Finally, we can link the object file `TEST.OBJ` to create an executable file (`OUTPUT.EXE`).

```
/* This is an example program for the 2171 containing STDIO */

#include  "stdio.h"

extern volatile int log_ana_port;

void main()
{

while (1)                    /* infinite loop for realtime system */
   {
   /* the code below clears the AR register (to assure error occurs),
      then tests a bit and calls error routine      */
   asm ("ar=0;  \
       af=tstbit 5 of ar;    \
       if eq call error_routine_; ");

   asm ("idle;");      /* wait for interrupt */
   }

}

void error_routine ()
{
   printf("Error Occurred");
```

```
   log_ana_port=0xDEAD;
}
```

Listing 22 ADSP-2171 STDIO.H Include Main Routine

```
/* This is an example program for the 2171 containing STDIO */
#include "stdio.h"

extern volatile int log_ana_port;

void main()
{

while (1)               /* infinite loop for realtime system */
  {
  /* the below code clears the AR register (to assure error
     occurs), then tests a bit and calls error routine */
  asm("ar=0; \
       af=tstbit 5 of ar;    \
       if eq call error_routine_;");

  asm("idle;");        /* wait for interrupt */
  }

}
void error_routine()
{
  printf("Error occurred");
  log_ana_port=0xDEAD;
}
```

Listing 23 ADSP-2171 Main Routine

```
.SYSTEM example_system;

.ADSP2171;
.MMAP0;

.SEG/ROM/BOOT=0                 boot_mem[2048];

.SEG/RAM/PM/ABS=0/CODE/DATA     int_pm[2048];
.SEG/RAM/DM/ABS=0x3000          int_dm[2048];
.port/dm/abs=0x2000             log_ana_port;
.ENDSYS;
```

Listing 24 ADSP-2171 Architecture Description File

```
bld21 -c 2171.sys
asm21 2171_hdr.dsp -c -s
g21 test.c -S -a 2171 -save-temps  -g -Wall
```

```
c:\adi_dsp\21xx\etc\cpp -P -undef -D__GNUC__=2 test.s test.is
c:\adi_dsp\21xx\etc\asm2 -c -s -o test test.is -2171
g21 -runhdr 2171_hdr.obj test -a 2171.ach -o output -map
```

Listing 25 ADSP-2171 Batch File

2. 21xx EZ-KIT-Lite Talkthru Shell Program

(C TIP - DSPatch #35: Anatomy of a 2100 family C program)

This C tip explains in detail a basic ADSP-2100 family C program. This C routine provides a simple shell for use with the ADDS-21XX-EZ-LITE. The EZ-LITE is the low-cost ADSP-2181 based evaluation board. While the EZ-LITE comes with software, it does not include the C compiler. So this C-tip assumes that you have the complete version of the Analog Devices software (ADDS-21XX-SW-PC). The current version of the software is release 6.0.

Our main C program is `CTIP35.C`. As you can see from the listing, individual lines have been numbered. The lines are explained in the corresponding sections below. `CTIP35.C` when run on the EZ-LITE takes the input from the AD1847 stereo codec called LEFT_IN and RIGHT_IN and loops it back to the 1847 by writing to LEFT_OUT and RIGHT_OUT. You can add on to this simple talk-thru program by modifying the inputs before writing to the outputs.

The file CTIP35.ZIP contains all the system files needed to test this application on your own EZ-LITE system. CTIP35.ZIP is available on the Analog Devices BBS or FTP. Files included in the ZIP file are:

CTIP35.C
SIGNAL.H,
2181_HDR.DSP
TALK_47.DSP
BUILD.BAT
2181.ACH

The file TALK_47.DSP is an assembly routine that is called by CTIP35.C and initializes the interface between the ADSP-2181 and the AD1847. The file SIGNAL.H is a header file that contains macros for handling interrupts. 2181_HDR.DSP contains a modified run-time header for the 2181. BUILD.BAT is a batch file that assembles 2181HDR.DSP and then invokes the C compiler and all its switches. And finally 2181.ACH is an architecture file that can be used with the EZ-LITE.

For more information on C coding with the Analog Devices' DSPs refer to the C tools manual, the ADDS-21XX-SW-PC release note, or previous C-tip articles.

CTIP35.C

```
   /* CTIP 35 */
   /* this program can be used as a shell for C programs running on the
      EZ-LITE board or as an basic example of C code */

3  #include <signal.h>
2  #define  DM_Wait_Reg            *(int *)        0x3ffe

5  extern init_1847();
3  void new_sample();

4  asm(".var/dm/ram/circ rx_buf_[3];             /* Status + L data + R data */\n"\
4      "\t.var/dm/ram/circ tx_buf_[3];           /* Cmd + L data + R data    */\n"\
4      "\t.init tx_buf_: 0xc002, 0x0000, 0x0000; /* Initially set MCE        */\n"\
4      "\t.global rx_buf_, tx_buf_;");

1  volatile int left_in, right_in;
1  volatile int left_out, right_out;
   int stat_flag;

   void main()
   {
5          init_1847();                      /* initialize 1847 interface */

4          asm("set fl1;");                  /* set LED on EZ-LITE */

2          DM_Wait_Reg=0x0fff;

3          interrupt(SIGSPORT0RECV, new_sample);

6          while(1){

6            asm("idle;");                   /* wait for an interrupt */

6            right_out=right_in;             /* loop back for talk-thru */
6            left_out=left_in;

           }
   }

3  void new_sample()
   {
4          asm("ena sec_reg;");
4          asm("ax0 = dm(rx_buf_+1);");      /* receive new sample */
4          asm("ax1 = dm(rx_buf_+2);");
4          asm("dm(left_in_) = ax0;");       /* write to memory */
4          asm("dm(right_in_) = ax1;");

4          asm("ax0 = dm(left_out_);");      /* read output sample */
4          asm("ax1 = dm(right_out_);");
4          asm("dm(tx_buf_+1) = ax0;");      /* transmit to codec */
4          asm("dm(tx_buf_+2) = ax1;");

4          asm("dis sec_reg;");
   }
```

Extensions to the ANSI C compiler

The Analog Devices GNU based C compiler conforms to the ANSI C standard. However, there are additions or extensions to the ANSI C language that are specific to the ADSP processors. The instruction:

```
volatile int left_in, right_in;
```

is an example of an extension to the int data type. Using volatile will force the DSP to place the variables `left_in` and `right_in` into memory locations -- as opposed to manipulating the data in registers only. Another extension to ANSI C is the pm data type. Using the data type:

```
int pm coeff[10];
```

will create a buffer in the DSP's program memory for storing coefficients. Without the pm extension all data will be placed in data memory.

Accessing memory-mapped registers in C

The DSP reserves a section of internal memory for configuration registers. Configuring the serial ports, internal timer, IDMA, BDMA, memory wait states, etc. is accomplished by writing to these memory mapped registers. To accomplish this in C, use a `#define` command to create a label or pointer for the memory location. In our CTIP35.C program we're initializing the wait states for the IO memory space using:

```
#define DM_Wait_Reg *(int *) 0x3ffe
```

Then in the code, you can write to that memory location using:

```
DM_Wait_Reg=0x0fff;
```

Handling interrupts

The DSP is an interrupt driven processor. In order to program those interrupts in C, it would be helpful if you understood what interrupts are available on your DSP of choice. The 2181 for example has interrupts for serial ports, external signals, timer, powerdown, etc. If you don't feel like cracking open the data sheet or user's manual, you can use the CTIP35.C as a shell and skip this section. If you want to understand how we've configured the interrupts or learn how to configure interrupts for your own system then read on.

The first step to handling interrupts is to include the header file SIGNAL.H in your program using the line:

```
#include <signal.h>
```

SIGNAL.H contains macros for each interrupt of each 2100 family processor.

Note: SIGNAL.H is included with your software tools. However, even if you have the latest software, you may not have the latest version of the file if you are using version 5.1. It is available for download from our BBS or FTP site or is included with the files if you download CTIP35.ZIP.

For CTIP35.C, we need to set up the interrupt for Serial Port 0 Receive. The macro has the following syntax:

```
interrupt(SIGSPORT0RECV, new_sample);
```

This line identifies the function `new_sample` as the interrupt handler for serial port 0 receive interrupts.

Embedding assembly instructions in C code

Admittedly, as soon as you add in-line assembly code you begin to sacrifice some of your C code's portability. However, there are just some instructions that you can only perform in assembly. Creating a buffer can be done just as easily in C as in assembly. However, only using the assembly directive:

```
asm(".var/dm/ram/circ rx_buf_[3];");
```

can you create a circular buffer. The IDLE instruction and system register accesses such as the IMASK also need to be done with assembly code. The general format for embedding an assembler instruction is:

```
asm("assembly code;");
```

To include several assembly lines of code, use the format:

```
asm("first instruction;" \
"second instruction;");
```

The backslash (\) has to be the last character on the line (including comments). You can also use \n and \t for creating line returns and tabs in you listing file.

Calling DSP programs/routines

Another way to add assembly code to your C routine is to place the DSP code in a C callable function. The program TALK_47.DSP contains code to initialize the 1847 codec. TALK_47.DSP contains the routine `init_1847()` which is called from C. An underscore must be added to labels and variables that are initialized in C and accessed in assembly. The DSP label in TALK_47.DSP is `init_1847_`.

Handling Real Time environments

Most DSP programs are based on external signals. This program waits until a Serial Port Receive interrupt occurs and then performs the loop-back. The WHILE instruction is used to create an infinite loop. The IDLE assembly instruction makes the processor wait for an interrupt. The catch is your code has to complete before the next interrupt would occur. For example, if you receive data from the 1847 every 16kHz you'd have 1/16kHz or 62.5 micro-seconds between interrupts. Using a 33Mhz 2181, this equates to 2062.5 DSP cycles.

BUILD.BAT

```
7  asm21 2181_hdr -c -2181
7  g21 -a 2181 -o ctip35 -I. -mreserved=i2,i3 -runhdr 2181_hdr.obj ctip35.c
     talk_47.dsp -g -save-temps -mlistm
```

Compiling your program

Before using the g21 command to compile the `CTIP35.C` program, you need to assemble your new 2181 run time header. Use the -c switch to make the `2181_HDR.OBJ` file case sensitive. The switches that are only needed for debug are the `-g`, `-save-temps`, and `-mlistm`.

After using the batch file to compile your C code, you can download the executable `CTIP35.EXE` to the EZ-LITE using the EZ-LITE monitor program that runs under windows.

3. EE-Notes Series - C Programming Guidelines For The ADSP-21xx

3.1 Accessing Memory-Mapped registers and Non-Memory-Mapped Registers In C

Contributed by Emmanuel Archer

This DSP EE-Note series has been conceived as guidelines for users programming our 21xx-DSPs in C. It consists of several pieces of examples as opposed to a whole project. These examples have been conceived general-purposed to allow the users to run them without any piece of hardware, just using the Simulator. The aim of these guidelines is namely to illustrate some fundamentals with appropriate examples, making thus the theory better understandable.

Covered issues are:

- Accessing memory-mapped registers and non-memory-mapped registers in C (EZ-Notes Number 01)
- Language extensions: Memory storage type, asm and inline constructs (EZ-Notes Number 02)

The chosen DSP to implement the examples is mostly the ADSP-2181. The code compatibility of the 21xx DSPs makes however most of the examples adaptable for any other processor of the Family.

The code samples for these DSP EZ-Notes are available to download from our BBS or FTP site under the file names C_EZxx.zip, where xx=Number of the corresponding EZ-Notes.

The file C_EZ01.zip contains the files needed to run the example(s). The files included are:

regs.c	*main C program*
cdef2181.h	*header file defining the memory-mapped registers*
regs.bat	*batch file to compile the example*
2181.ach	*architecture file for the ADSP-2181*

Printed codes (here. reg.c) have every 5th line numbered, so to better reference the further comments made in the text section of these Notes

For more information on C coding with the Analog Devices' 21xx-DSPs refer to the C tools Manual, the ADDS-21XX-SW-PC Release Notes and other EZ-Notes of this series.

Accessing memory-mapped registers in C

Memory-mapped registers reside in a region of internal reserved data memory of the DSP. These registers are used to configure features like the internal timer, the serial ports, memory wait states, programmable flags, etc.
The header file cdef2181.h uses the `#define` preprocessor directive to assign a symbolic name to the address of each memory-mapped register.of the ADSP-2181. This header file will be used by most of the examples throughout these notes. The symbolic names used are the same as provided in the file def2181.h included with the development software. For ease of referencing these symbolic labels the file cdef2181 is printed out in these notes.

The command:

```
#define  Prog_Flag_Data  *(int *)  0x3fe5
```

creates a label for the memory location DM(0x3fe5) by casting 0x3fe5 as a pointer to an integer and then de-referencing it. This means that the symbol Prog_Flag_Data becomes equivalent to the integer value pointed to by 0x3fe5. The C statement

```
Prog_Flag_Data=0x0000;
```

writes the value 0 to the memory location 0x3fe5 and is thus equivalent to the assembly language instructions:

```
AX0=0x0000;
DM(0x3fe5)=AX0;
```

CDEF2181.H

```
/* cdef2181.h */
/* header file assigning the ADSP-2181's memory-mapped registers */
/* to user-defined names */

#define  Sys_Ctrl_Reg                              *(int *)    0x3fff
#define  Dm_Wait_Reg                               *(int *)    0x3ffe
#define  Tperiod_Reg                               *(int *)    0x3ffd
#define  Tcount_Reg                                *(int *)    0x3ffc
#define  Tscale_Reg                                *(int *)    0x3ffb
#define  Sport0_Rx_Words1                          *(int *)    0x3ffa
#define  Sport0_Rx_Words0                          *(int *)    0x3ff9
#define  Sport0_Tx_Words1                          *(int *)    0x3ff8
#define  Sport0_Tx_Words0                          *(int *)    0x3ff7
#define  Sport0_Ctrl_Reg                           *(int *)    0x3ff6
#define  Sport0_Sclkdiv                            *(int *)    0x3ff5
#define  Sport0_Rfsdiv                             *(int *)    0x3ff4
#define  Sport0_Autobuf_Ctrl                       *(int *)    0x3ff3
#define  Sport1_Ctrl_Reg                           *(int *)    0x3ff2
#define  Sport1_Sclkdiv                            *(int *)    0x3ff1
#define  Sport1_Rfsdiv                             *(int *)    0x3ff0
#define  Sport1_Autobuf_Ctrl                       *(int *)    0x3fef
#define  Prog_Flag_Comp_Sel_Ctrl                   *(int *)    0x3fe6
#define  Prog_Flag_Data                            *(int *)    0x3fe5
#define  BDMA_Word_Count                           *(int *)    0x3fe4
#define  BDMA_Control                              *(int *)    0x3fe3
#define  BDMA_External_Address                     *(int *)    0x3fe2
#define  BDMA_Internal_Address                     *(int *)    0x3fe1
#define  IDMA_Control                              *(int *)    0x3fe0
```

The following demo program `regs.c` illustrate the use of the header file cdef2181.h for accessing memory-mapped registers. It consists of two functions: The first one sets the timer of the 2181, which can be used to set a timer interrupt (See EZ Notes Number 05). The other one programs one programmable flag of the 2181 in the Flags Registers to generate a software reset of the EZ-Kit Lite

REGS.C

```
      /* regs.c */

      /* demo program showing accesses to memory-mapped
      and non-memory-mapped registers */

1     #include "cdef2181.h"

      void set_timer(void);                    /* prototyping the used functions */
      void reset_kit(void);

      void main(void)
5     {
      set_timer();

      reset_kit();

      asm ("dis timer;");                      /* disabling the timer */
10    }


      /* The routine reset_kit performs a software reset on the EZ-KIT LITE   */
      /* Requirement: connect the PF0 pin to the reset circuitry              */
      /* with a 0-Ohm resistor on the R28 location.                           */

      /* This is realised through the programming of the PF0 flag and         */
15    /* represents thus at the same time a general model of how dealing      */
      /* with the PF flags.                                                   */

      void reset_kit(void)
      {
      Prog_Flag_Comp_Sel_Ctrl=0x7b01;                     /* PF0=output */

20    Prog_Flag_Data=0;                                   /* PF0 on low */


}
      void set_timer(void)
      {
      asm(”ifc=0xff;                           /* clear pending interrupts   */ \
25    nop;”);                                  /* delay for ifc writing      */

      Tscale_Reg =1;
      Tcount_Reg =2000;
      Tperiod_Reg =2000;

30    asm("imask=1;                            /* unmask timer interrupt     */ \
      ena timer;");                            /* timer starts               */
      }
```

Line 1: header file

The `#include ”cdef2181.h”` C preprocessor command is used to establish access to the file cdef2181.h. This file is a *user* header file -as opposed to a *system* header file that interfaces to the operating system- that contains the memory-mapped registers declarations used by the C program.

Lines 19-20 & 26-28: register accesses

Lines 19-20 program the flag PF0 by writing the memory-mapped Flag Control and Data registers with the corresponding values.
Lines 26-28 configure the internal timer by loading the memory-mapped Timer registers with the desired values.

Accessing non-memory-mapped registers in C

Non-memory-mapped registers are dedicated DSP registers such as IFC, IMASK, ASTAT,etc.
Accessing these system registers can only be done with assembly code. The demo program regs.c shows some examples of embedding assembly instructions in C code:

Lines 24 & 30
These instructions make use of the asm() construct

```
asm("assembly instruction;");
```

to respectively access the system registers IFC and IMASK.

The backslash (\) is used as a line continuation to embed more than one assembly instruction in a single asm() string. It allows the use of separate lines for the different instructions making thus the code easier to read and comment. The backslash must be the last character on the line including comments.
The following batch file can be used to compile the C code:

REGS.BAT

```
g21 regs.c -a 2181 -g -o regs -save-temps -mlistm -v -Wall
```

- This command line controls the operations of other software development tools as it performs the translation process (C preprocessing, compiling, assembling and linking) to create the executable file regs.exe. Some other performed operations include allocating memory for the stack and adding a runtime header. The verbose switch (`-v`) lets the user monitor all these operations by making the compiler print the commands issued to execute each stage.

- Because no runtime header file is specified, the compiler uses the .ADSP2181 directive of the architecture file 2181.ach to determine which ADSP21xx predefines to pass (= which runtime header file to use).

- `-a 2181` specifies the system architecture file 2181.ach

- `-o regs` select the name regs.* for the final output files

The -g, -save-temps, and -mlistm switches are only used for debugging.:

- `-g -save-temps` are needed for using the simulator's C source debugger (CBUG)

- `-mlistm` directs the compiler to output a merge list file. You must use the `-save-temps` command line switch with the `-mlistm` switch to see C and assembled listings

- `-Wall` echoes all recommended warning messages to the screen.

3.2 Language Extensions: Memory Storage Types, ASM & Inline Constructs

Contributed by Emmanuel Archer

Language extensions

The G21 C compiler supports a set of extensions to the ANSI standard for the C programming language. These extensions are specific to the ADSP processors and comprise:

- Support for separate program and data memory (keywords `pm, dm`)
- Support for `inline` functions
- Support for `asm()` inlining of assembly language

The following demo code will help illustrate the use of these extensions and provide further information on them.

LANG_EXT.C

```
/* lang_ext.c */
/* program demonstrating the use of the G21 extensions */
/* to the C language */


1     #include "pointdef.h"              /* contains predefined pointers */

      int aux;

      static int dm x ;                  /* placing variables in specific */
      static int dm y ;                  /* memory spaces */
5     static int pm z ;
      static int dm i[3]={10,20,30};

      DINT pointer;                      /* DINT is defined in pointdef.h */
                                         /* for a pointer to a dm integer */

      static inline int add(void)        /* define add() as inline */
      {
10    z=x+y;
      return z;
      }

      static inline void Set_Fl(short flag)     /* define Set_Fl() as inline */
                                                /* this function just sets */
                                                /* Flag 0, 1 or 2 */
      {
15    switch (flag)
            {
            case 0:     asm volatile("set fl0;");
                        break;
            case 1: asm volatile("set fl1;");
20                      break;
```

```
            case 2: asm volatile("set fl2;");
                        break;
            }
      }

25    static inline void reset_fl1(void)        /* to reset Flag 1 and count */
                                                /* the amount of resets */
      {
      static int pm rst_fl1_count;
      rst_fl1_count++;
      asm volatile("reset fl1;");
30    }

      void main (void)
      {                               /* several pointer operations */
                                      /* based on the use of the keywords */
                                      /* pm/dm and the file pointdef.h */

      pointer=i;                      /* pointer points to i[0]=10 */
      x = *pointer++;                 /* x=10 ; pointer points to i[1]=20 */
35    aux=(*pointer)++;               /* aux=20; i[1]=21; */
      y=i[1];                         /* y=21 */
      add();

      Set_Fl(0);                      /* calling the inline functions */
      asm volatile("reset fl0;");

40    Set_Fl(1);
      reset_fl1();

      asm volatile("toggle fl1;");
      reset_fl1();
      }
```

Dual Memory Support (pm dm)

The two keywords (pm, dm) support the dual memory space (separate program and data memory spaces where program memory can store both data and opcodes and data memory stores data), modified Harvard architecture of the ADSP-2100 Family processors, as opposed to the Von Neumann architecture where program instructions and data are all stored in one common memory space.

The keywords (`pm`, `dm`) are used to specify the location of a static (or global) variable. Being thus able to explicitly access PM data locations (e.g. for fetching filter coefficients) and DM data locations, you can take advantage of the DSP's architecture where two data words and an instruction can be fetched in a single cycle.

Lines 3-6
These statements illustrate the placement of variables in PM or DM spaces.

Some rules applying to the use of the dual memory support keywords are:

- Without specification of pm/dm, G21 uses data memory as the default memory space. For example the variable `aux` **(line 2)** is placed into data memory.

- Program memory is the dedicated memory space for functions

- Automatic variables (= local variables within a function) are placed onto the stack, which resides in data memory. To be able to use the pm keyword inside a function, you have to give the variable the storage class `static` by using the qualifier of the same name. **(line 27)**

Pointer declarations using pm/dm

A further usage of the dual memory support keywords is qualifying pointer declarations within the dual memory space. For instance, the following statement

```
int dm * pm pd;
```

declares `pd` as a pointer residing in program memory and pointing to a data memory integer.

Our demo program lang_ext.c uses this pointer declaration feature, too, by including **(line 1)** the user header file `pointdef.h`, which contains some predefined pointer types. This file can be seen as a shell that can be customized to be used in any code dealing with pointers.

POINTDEF.H

```
typedef int pm *  PINT;                 /* PINT defines a type which is a pointer
                                           to an integer which resides in pm */

typedef int *  DINT;                    /* DINT defines a type which is a pointer
                                           to an integer which resides in dm */
                                        /* other syntax: typedef int dm *  DINT; */

typedef float pm *  PFLOAT;             /* PFLOAT defines a type which is a pointer
                                           to a float which resides in pm */

typedef float *  DFLOAT;                /* DFLOAT defines a type which is a pointer
                                           to a float which resides in dm */
                                        /* also: typedef float dm *  DFLOAT; */
```

Line 7 of the C program

```
DINT pointer;
```

uses for example the predefined pointer type DINT of the file pointdef.h to declare the variable `pointer` as a pointer residing in dm (per default) and pointing to an integer which resides in dm.

One could wonder: why not also defining the memory space (pm/dm) of the pointer within the typedef statement?

It is not possible to do so.The compiler would for instance flag an error message on this construct:

```
typedef int * pm DINTP;     /* trying to define a type which is a pm pointer
                               pointing to a dm integer */
```

The pointer storage type must be defined by the programmer, i.e. :

```
#include "pointdef.h"
DINTD pm dd1;                    /* = int * pm dd1 */
                                 /* dd1 is a pm pointer pointing to a dm integer */
```

Lines 33-36 of the C program shows some pointer operations that illustrate the handling with variables and pointer explicitly declared to be in pm/dm.

To finish with this section on pointer declarations, it could be added that, since functions always reside in program memory, pointers to functions always point to program memory.

Inline Function Support Keyword (inline)

The G21 `inline` keyword directs G21 to integrate the code for the function declared as inline into the code of its callers. This makes execution faster by eliminating the function-call overhead.

Lines 8, 13 and 25 of our C example show examples of function definitions that use the `inline` keyword.

Note that the definitions of the functions precede the calls to the functions. This is to prevent the calls from not being integrated into the caller.

How inlining functions affect code and execution can be seen under the simulator in the C code window (CBUG) and the assembly code window (Program Memory Window).

An alternative to explicitly declaring functions as inline is to use the switch `-finline-functions` when compiling your C code. This switch forces function inlining by directing the compiler to integrate all simple functions into their callers. Which functions are simple is decided by the compiler.

Inline Assembly language Support Keyword (asm)

The asm() construct allows the coding of assembly language instructions within a C function and is thus useful for expressing assembly language statements that cannot be expressed easily or efficiently with C constructs. Some examples are:

- `asm(" ifc=0xff ")` /* accessing non-memory-mapped registers can only be done in assembly language */
  /* *Please refer to EZ NOTES Number 01* */

- `asm volatile ("set fl0;");` /* **line 17** of lang_ext.c */
  /* `set fl0` is a processor specific instruction
  that can only be expressed in assembly language */

The extension `volatile` is used to prevent the asm() instruction from being deleted or moved significantly.

- A particular register variable can be assigned using asm(). The following statement assigns the register AX0 to the C variable x:

```
register int x asm("AX0");
```

- A more complex and flexible form of the asm() construct that allows to specify the operands of the assembly instruction using C expressions will be discussed in the next *EZ Notes (Number 03)*, whose subject is *Interfacing C and Assembly Language*

The following batch file has been used to compile the demo program:

LANG_EXT.BAT

```
g21 lang_ext.c -a lang_ext -g -v -Wall -save-temps -mlistm -fkeep-inline-functions -o
lang_ext -map
```

- The `-map` switch directs the linker to generate a memory map file of all symbols. This is useful for debugging purposes.

- The `-fkeep-inline-functions` (force keeping of inlined functions) switch directs the compiler to output a separate runtime callable version of the inlined functions. This switch is necessary, for instance if you want to set breakpoints within the functions, because it makes G21 output own assembler code for the functions (what the C compiler does not do otherwise).

After compiling the example, you can run it using the C Debugger of the simulator (CBUG). For information on how to use CBUG, please refer to the C Tools Manual and the Release Notes.

3.3 I/O Space for the ADSP-218x DSPs - Piping a C variable to an I/O Port

If you want to send the value of a C variable to an I/O port mapped in I/O memory space, use in-line assembly as shown in Listing below:

```
int myvar;
/* variable declared globally  */
void main(void     )
{
   myvar=256 ;
   asm(".external  myvar_;");
   asm("ax0=dm(myvar_);");
   asm("IO(0x0100)=ax0;");
}
```

ADSP-218x C Code For Variable I/O Space Access

The code in Listing 8 pipes the value of a C variable myvar to I/O port at address 0x0100. Note that the variable is declared globally to make it accessible to the assembly code. It is also necessary to declare the variable as an external before trying to access it. Finally, note the choice of the ax0 register to store the value. This register is regarded as a scratch register by the C compiler and is hence safe to use.

You can similarly use in-line assembly to direct a value from an I/O port into a C variable.

4. DSPatch C Programming Q & A's

1. Boot Paging In C

Q. Is there any way I can create a number of C programs and store them as different boot pages in a boot EPROM? I notice that there is an assembler directive for specifying the boot page but I didn't see anything similar in the C compiler.

A.
The ADSP-2101, ADSP-2105, ADSP-2111 and ADSP-21msp50 have the capability of booting in any one of eight pages of code from a single byte-wide external EPROM such as a 27512. To prepare C programs that will generate code to reside in different pages of the EPROM, follow this procedure:

1) A unique runtime header must go with each C program that is to reside in the EPROM. The standard runtime header supplied with the development software is shown below (with one addition, the BOOT=0 module qualifier):

```
.MODULE/BOOT=0                ADSP2101_Runtime_Header;

.EXTERNAL    main_, ____top_of_ram, ____top_of_stack;
.EXTERNAL    __lib_setup_heap, __lib_setup_errno, __lib_setup_exit;
.EXTERNAL    __lib_setup_interrupts_2101;
.EXTERNAL    __lib_restart_vector
.ENTRY       __lib_code_start;

__lib_code_start:
        IMASK=0;      {Disable all interrupts}
        L0=0; L1=0; L2=0; L3=0;           {All length registers }
        L4=0; L5=0; L6=0; L7=0;           { should be set to 0}
        M0=0; M1=1; M2=0; M3=-1;          {Modify registers set }
        M5=1; M6=0; M7=-1;                { to various constants}
        M4=^____top_of_stack;             {Frame pointer=top of stack}
        I4=^____top_of_stack;             {Stack pointer=top of stack}
        CALL  __lib_setup_heap;
        CALL  __lib_setup_interrupts_2101;
        CALL  __lib_setup_errno;
        CALL  __lib_setup_exit;
        CALL  main_;
__lib_code_hang:
         JUMP  __lib_code_hang;
.ENDMOD;
```

The runtime header is found in the directory ADI_DSP\LIB\SRC. The names of the runtime header files for the various ADSP-2100 family processors are:

2105_HDR.DSP
2101_HDR.DSP
2111_HDR.DSP
2150_HDR.DSP

The first C program, the program to reside in boot page 0, can use the standard runtime header supplied with the /BOOT=0 qualifier added to the .MODULE directive. The linker will default to the use of this

standard runtime header. For any additional C programs, the runtime header shown above should be copied into another file that we will call HDR1.DSP in this example. Since each runtime header must have a unique name, the module name should be changed. We will simply append a '1' to the module name as follows:

```
.MODULE/BOOT=1    ADSP2101_Runtime_Header1;
```

(Note that the /BOOT=1 qualifier is used to locate this C program on boot page 1.) Each module can have a different number appended to the name. This approach is also used in the next four lines of the runtime header as shown:

```
.EXTERNAL  main1_, ____top_of_ram, ____top_of_stack;
.ENTRY     __lib_code_start1;

__lib_code_start1:
        CALL  main1_;
```

With this approach, any number of additional runtime header files can be created.

2) Assemble the runtime header programs one at a time, including the standard one (for boot page 0), using the case sensitivity switch as follows:

```
ASM21 2101_HDR.DSP -c
ASM21 HDR1 -c
etc.
```

3) Create the C programs for each boot page and use the following names for each main procedure:

```
main(), main1(),
main2(), main3(),
etc.
```

4) Compile the C programs one at a time, using the boot page switch as shown below. Each runtime header's .MODULE statement must specify (with the /BOOT=n qualifier) the same boot page as the compiler's -bn switch.

```
CC21  prog0  -b0
CC21  prog1  -b1
CC21  prog2  -b2
etc.
```

5) Link all the compiled C programs with the assembled runtime header files as follows:

LD21 prog0 prog1 prog2 HDR1 HDR2 -e C_CODE -a C_SYS -c -lib

Note that the C program called PROG0 will be linked with the default runtime header file included with the development software. Also, the architecture description file must have boot pages declared.

2. C Compiler Command Line

Q. For my application, I specify many files on my C compiler command line. Sometimes, I run into the DOS limit of 128 characters on the line. Is there a more efficient way to list input files for G21/G21k?

A. The best way to specify a long list of files with the G21k or G21 compiler is through the use of the command line option @filename (undocumented feature!). This feature lets you specify (on the command line) a file containing a list of files to send to the compiler. The format for the file used in the command line, filename, requires an ASCII file with one path\filename per line. The following example shows a G21 command line that uses the files listed in files_all as input files:

```
g21 @files_all -a 21.ach
```

3. G21 Problems with Multiplication Results

Q. I am having a problem with a routine written for the ADSP-2101 when I call it from a C program. When I simulate the assembly language routine, or use it with other assembly routines, it works fine. However, when I call the assembly language routine from a C program, the multiplication results aren't what I expect.

A. The ADSP-2101 performs multiplications in one of two modes--fractional mode and integer mode. The mode is controlled by bit 4 in the MSTAT register. In fractional mode, the processor shifts the multiplier result left one bit before writing to the result register. This shift eliminates the extra sign bit generated when the two input operands are fractional signed numbers (frequently referred to as 1.15 format). In the integer mode, the left shift doesn't occur. At reset, the ADSP-2101 defaults to the fractional mode.

The C compiler works in integer mode and changes the processor's default mode. You must change the multiplier to fractional mode before performing any multiplications. Insert a DIS M_MODE; instruction at the beginning of your assembly routine to change the mode (clears bit 4 in the MSTAT register). Insert an ENA M_MODE; instruction at the end of your routine, before returning to the C program. The ENA M_MODE; instruction sets bit 4 of the MSTAT register and changes the multiplier back to integer mode.

4. Accessing Absolute Memory Locations in C

Q. I'm using the C compiler to develop code for an ADSP-2101 system. How can I store values in absolute locations in order to configure the memory-mapped control registers,or to write data to a peripheral residing in external memory?

A.
The code segment below illustrates an easy method to write data to absolute addresses:

```
#define DA_Conv *(int *) 0x2000

/* now define the mem-mapped registers for the ADSP-2101's internal timer */

#define tperiod *(int *) 0x3FFD
```

```
#define tcount *(int *) 0x3FFC
#define tscale *(int *) 0x3FFB

main()
{
/* write 0x100 to the D/A */
DA_Conv=0x100;

tperiod=300;
tcount=0;
tscale=300;
}
```

5. Using ADSP-2100 DM-Mapped Ports in C Programs

Q. I have a C program that I would like to use for the ADSP-2100. Specifically, I need to know how to take advantage of I/O ports connected to the ADSP-2100 using my C program.

A.
Any symbolic reference that can be made with ADSP-2100 assembly language can also be made with C language programs, including the reference to labels for ports.

When the C compiler for the ADSP-2100 operates, all labels used in the C program have an underscore appended to them in the ADSP-2100 assembly language output. Therefore, the key to referencing I/O ports is to name I/O ports (in ADSP-2100 System Builder and Assembler modules) with an appended underscore and then use the reference (in your C program) without the underscore. The following example of a DAC reference in a C program shows this.

```
extern  DA_OUTPUT;

main()
{
      int  i;
      for (i=0; i<10; i++)
       DA_OUTPUT = i;
}
```

In the C program DA_OUTPUT is declared external, has no appended underscore and is uppercase. In order to properly declare the port some in-line assembly code must be inserted with the `asm()` construct. Since the `asm()` construct only works within a function, a dummy function must be defined as follows:

```
dummy()
{
asm(".port   DA_OUTPUT_;");
asm(".global DA_OUTPUT_;)");
}
```

The D/A port in this example is actually named `DA_OUTPUT_.` Note that the port is uppercase with an underscore appended to it. When the C program is compiled, an underscore is appended to the `DA_OUTPUT` name.

6. Using Interrupts with C Programs for the ADSP-2100

Q. How can I handle interrupts when using C programs for the ADSP-2100?

A.
Interrupts can be handled in an ADSP-2100 C program by following these simple steps. First, the Run time Header (RTH) must be edited to include the interrupt vector or vectors. The RTI instructions in the standard run time header should be changed to jump instructions where the destination of the jump is the first instruction of the interrupt routine. If the jump is to a C program, the address label should be appended with an underscore (_). For example, if the jump is to a routine called "func" then the jump statement in the run time header should be "JUMP func_."

The interrupt or interrupts must also be properly enabled in the run time header. The setting of the interrupt control register (ICNTL) and interrupt mask register (IMASK) must be done by adding instructions to the run time header.

An example is shown to clarify the use of interrupts. This is an example of an edited run time header module.

```
.MODULE/ABS=0/RAM RTH;
.EXTERNAL       main_;
.EXTERNAL        ____top_of_ram;
.EXTERNAL       int_rtn_;

            RTI;
            RTI;
            RTI;
            JUMP    int_rtn_;

top:    M4=^____top_of_ram;
            I4=^____top_of_ram - 1;
            L0=0;
            L1=0;
            L2=0;
            L3=0;
            L4=0;
            L5=0;
            L6=0;
            L7=0;
            M0=0;
            M1=1;
            M2=0;
            M3=-1;
            M5=1;
            M7=-1;

            ICNTL=h#0F;
            IMASK=h#08;

            CALL main_;
            TRAP;
.ENDMOD;
```

The run time header file must be assembled using the case sensitivity switch "-c" so that capital letters and small letters will be treated as the same character.

The following C code example shows a main routine which is to be interrupted.

```
main()

{
      int i,k;
top:
      for (i=0; i<1000; i++)
            k=i;
      goto top;
}
```

The loop will be repeated until an interrupt occurs. The program flow will be directed to the interrupt routine and, after the interrupt routine is finished, will be redirected back into the main loop. An example of the interrupt routine program is shown below:

```
int_rtn()

{

asm(" MX0=5;  ");
asm(" MY0=10 ");
asm(" MR=MX0*MY0(SS); ");
asm(" DM(i4,m7)=MR1; ");
asm(" RTI; ");

}
```

The C programs must be compiled and then linked. When linking, the -c switch must be used in order to create the run time stack correctly. An example of the link command is shown:

```
G21 run_hdr c_func1 c_func2 -e my_prog -x -g -c -a my_sys
```

7. Problems with the C Preprocessor

Q. I have a question about the proper usage of the .include statement in my ADSP-2101 program. In my assembly module, when I use the following statement:

```
.include
<c:\adi_dsp\21xx\include\def2101.h>;
```

I get the following error message:

```
asm21:
"c:\dsp\21xx\include\def2101.h",
line 0: illegal lhs '#'.
```

What's the problem? Doesn't the C pre-processor recognize #define statements in include files?

A. When you use the C compiler to assemble ADSP-2100 Family code, the C pre-processor is invoked. The problem you are seeing is caused by the order in which things happen when you assemble your code. During assembly, two pre-processor calls are made: one for the C pre-processor and one for the Assembly pre-processor. The C pre-processor is called first. Since the .include directive is an assembler directive, the C pre-processor ignores it. Then, when the Assembly pre-processor runs, it inserts the .include file. The file you have specified def2101.h contains a directive that is not proper assembly syntax, but C language syntax. The Assembly pre-processor doesn't recognize the #define instructions contained in the definition header file.

The solution is to always use the following C directive:

```
#include <filename>
```

when you use an include file. This ensures that all C-style commands are interpreted correctly.